Predictive Context-sensitive Fuzzing

Pietro Borrello*, Andrea Fioraldi†, Daniele Cono D'Elia*,
Davide Balzarotti†, Leonardo Querzoni* and Cristiano Giuffrida‡
*Sapienza University of Rome
†EURECOM
‡Vrije Universiteit Amsterdam
{borrello, delia, querzoni}@diag.uniroma1.it, {fioraldi, balzarot}@eurecom.fr, giuffrida@cs.vu.nl

***Abstract*—Coverage-guided fuzzers expose bugs by progressively mutating testcases to drive execution to new program locations. Code coverage is currently the most effective and popular exploration feedback. For several bugs, though, also how execution reaches a buggy program location may matter: for those, only tracking what code a testcase exercises may lead fuzzers to overlook interesting program states. Unfortunately, context-sensitive coverage tracking comes with an inherent state explosion problem. Existing attempts to implement context-sensitive coverage-guided fuzzers struggle with it, experiencing non-trivial issues for precision (due to coverage collisions) and performance (due to context tracking and queue/map explosion).**

In this paper, we show that a much more effective approach to context-sensitive fuzzing is possible. First, we propose function cloning as a backward-compatible instrumentation primitive to enable precise (i.e., collision-free) context-sensitive coverage tracking. Then, to tame the state explosion problem, we argue to account for contextual information only when a fuzzer explores contexts selected as promising. We propose a prediction scheme to identify one pool of such contexts: we analyze the data-flow diversity of the incoming argument values at call sites, exposing to the fuzzer a contextually refined clone of the callee if the latter sees incoming abstract objects that its uses at other sites do not.

Our work shows that, by applying function cloning to program regions that we predict to benefit from context-sensitivity, we can overcome the aforementioned issues while preserving, and even improving, fuzzing effectiveness. On the FuzzBench suite, our approach largely outperforms state-of-the-art coverage-guided fuzzing embodiments, unveiling more and different bugs without incurring explosion or other apparent inefficiencies. On these heavily tested subjects, we also found 8 enduring security issues in 5 of them, with 6 CVE identifiers issued.

I. Introduction

Fuzz testing (or fuzzing for short) techniques earned a prominent place in the software security research landscape over the last decade. Their efficacy in generating unexpected or invalid inputs that make a program crash helps developers catch bugs early, even before they turn into vulnerabilities [1]. As an example, their deployment at scale in the OSS-Fuzz [2] initiative has led so far to the discovery of over 30 000 bugs in the daily testing of hundreds of open-source projects.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24113
www.ndss-symposium.org

The most popular and researched form of fuzzing is coverage-guided fuzzing (CGF), which uses code or other coverage information from program execution to deem whether the current testing input led to *interesting* (for example, previously unseen) portions of a program. The main intuition behind much CGF research is that code coverage is strongly correlated with bug coverage [3] and no dynamic testing technique can detect a bug if execution does not reach the corresponding program point at least once. A flourishing topic of research is to enlarge the covered code by improving the effectiveness of the input generation process, e.g., by guiding input mutations to meet complex control-flow conditions in the program [4], [5], [6].

However, for software testing, coverage is only one part of the equation [7], and the ultimate metric for the effectiveness of fuzzing remains the ability to discover bugs. As recently observed in [8], successful CGF embodiments balance between exploration and exploitation. While exploration aims to increase coverage, exploitation tries to trigger bugs in already-covered program regions by varying the inputs used to reach them before. As there is no immediate feedback for exploitation, fuzzers have to count on input mutations to execute such code "sufficiently well" to trigger bugs in it [8].

Therefore, other efforts focus on retaining for further mutation inputs that, while being equivalent to prior executions in terms of covered program points, exercise new valuable execution paths and/or internal states of the program [9]. Intuitively, these inputs offer alternative (and possibly more profitable) "starting points" for the above-said mutations to trigger some bugs. For example, most state-of-the-art CGF systems track edge coverage information to distinguish visits to the same basic block from different predecessor blocks [10].

Edge coverage and other *function-local* metrics track and summarize program execution for its effects on entities (e.g., code blocks, variable values) involving individual functions. A limitation of this strategy is that it may lead a fuzzer to overlook internal program states for which also *how* an entity is reached matters. In program analysis, this concept goes under the name of *context-sensitivity* and has seen many applications, such as refining the precision of pointer analyses [11] and developing compiler optimizations [12].

ANGORA [1] showcases the benefits of context-sensitivity for fuzzing by augmenting edge coverage with *calling-context* information, which captures the sequence of active function calls on the stack leading to the currently executing function [13]. In principle, such a *fully context-sensitive* approach can differentiate the coverage of each testcase in a fine-grained manner and lead to the discovery of more bugs [1], [10].

However, as an accurate call-stack tracking and context encoding would be costly and degrade the fuzzer's throughput, ANGORA [1] and other fuzzers [14], [15] embody a *best-effort* strategy for full context-sensitivity. In particular, they model the calling context as a hash of the call stack and compute context-sensitive coverage identifiers by combining the hash for the current context with the function-local edge identifier upon entering a basic block. This scheme is naturally prone to collisions, which are detrimental to fuzzing as they may lead to missing many relevant testcases [16]. To mitigate this shortcoming, these fuzzers employ larger coverage maps (e.g., 2^{20} entries in ANGORA [1]), a choice that does not come cheap as it can severely harm the fuzzing throughput.

More importantly, as we study, fully context-sensitive approaches are prone to state explosion, enlarging the fuzzer's queue with additional testcases that further reduce fuzzing efficiency, as the fuzzer will often fall short of the time needed to schedule or sufficiently mutate them [10].

In this paper, we will refer to all such kinds of detrimental effects as the *internal wastage* that the fuzzer experiences.

Our approach: We argue that the current "all-or-nothing" approach to context-sensitive fuzzing is unnecessarily inefficient, and that a much more effective approach is possible. The design we propose builds on three main insights:

❶ We show that we can do away with run-time call stack tracking by relying on a code specialization primitive. For a given calling context, with *function cloning* we create a clone of each callee and redirect the caller invocation to it. As a result, existing function-local coverage tracking techniques can naturally disambiguate calling contexts with no changes. For example, edges from cloned functions can benefit from the collision-free encoding of modern fuzzers as their presence implicitly carries (precise) contextual information, opposed to current approaches that enforce (and, as we study, further deteriorate) an imprecise hash-based edge encoding scheme.

❷ We show that, while fully context-sensitive approaches are in general problematic due to an inherent state explosion problem, *selective* approaches can be a much better alternative. Through techniques that restrict cloning to program portions that are likely to benefit from contextually refined edge profiles, we can bound our cloning efforts to trade a modest increase in program size with efficient context-sensitivity provided only for the callees that "*matter*". We term our approach *predictive context-sensitive fuzzing*.

❸ We show that data-flows for function call arguments can be an effective predictor for several such regions. We analyze the flow of objects through function arguments at call sites and pick those call targets that see a highly *diverse* incoming data-flow if compared to other invocations of the function in the rest of the program. The intuition is that such differences may reflect relevant variations in program behavior that we want to capture by means of context-sensitive coverage tracking. Moreover, we show how to realize the strategy without analyzing full calling contexts, but building instead atop a standard context-insensitive inter-procedural analysis.

This design results in a practical and performant context-sensitive fuzzing solution. On the popular FuzzBench suite [17], our approach can reveal more unique bugs than ANGORA-style context-sensitivity ($+22.55\%$). Also, it outperforms a collision-free edge coverage solution boosted with link-time optimization ($+11.6\%$), with the bugs found across trials being different than with edge coverage alone by 19.2%.

These improvements mainly come from our ability to trigger bugs in code regions that other solutions explore but fail to exploit. Our approach experiences only a limited growth of retained testcases ($+26\%$ w.r.t. edge coverage, opposed to $+81.7\%$ from ANGORA-style context-sensitivity) and a modest impact on the fuzzing throughput (-6.5% vs. -20.3%).

Finally, despite the FuzzBench subjects we study are well-tested in prior efforts and daily in OSS-Fuzz, our tests revealed 8 long-standing security issues involving 5 of these subjects, with 6 CVE identifiers issued upon responsible disclosure.

Contributions: To summarize, this paper proposes:

- A selective approach to context-sensitive fuzzing that augments only promising program portions with contextual information, using function cloning to enable a collision-free encoding with no run-time tracking machinery;
- A data-flow analysis to predict program portions likely to benefit from contextual refinement when fuzzing, using a strong signal given by call-argument value diversity among the different callers of a given target function.
- An open-source implementation in LLVM that produces programs suitable for out-of-the-box fuzzing (available at: https://github.com/eurecom-s3/predictive-cs-fuzzing).
- An evaluation of our approach atop AFL++ on the FuzzBench suite, where we consistently outrank state-of-the-art context sensitive and insensitive techniques, also exposing 8 enduring vulnerabilities in 5 popular subjects.

II. BACKGROUND

This section covers fundamental concepts of fuzzing and the points-to analysis primitives that back our predictive context-sensitive approach.

A. Coverage-guided Fuzzing

Fuzzing techniques have a prominent place in software security research due to their effectiveness in bug discovery [18]. In the most naive embodiment, a fuzzer is a system that attempts repeated executions of a target program over randomly generated testcases while monitoring it for crashes. Many techniques are available to optimize the testcase generation process, e.g., to discover more bugs within a given time budget [19] or prioritize specific code regions for testing [20].

The amount of information that a modern fuzzer acquires during the (many) executions of the program under test can vary, leading to a distinction between black-box [21], [22], white-box [23], [24], and grey-box [25], [26] fuzzers. In particular, grey-box fuzzers use lightweight instrumentation to track coarse-grained state information such as the code coverage achieved by each testcase and are largely popular due to their effectiveness.

As we anticipated in Section I, tracking code coverage can also serve as a *feedback* for coverage-guided fuzzers, allowing them to distinguish the program behaviors distinctive of each testcase by profiling, e.g., the control-flow edges taken during

the execution (edge coverage). Ultimately, this choice improves the ability of a fuzzer to find vulnerabilities [27].

Coverage-guided fuzzers instrument program code to update a *coverage map* (e.g., when the program takes a control-flow edge) that eventually serves as a profile of the testcase execution. Some also keep track of hit counts at coverage points. A relevant aspect of map updates involves *collisions*, which harm the effectiveness of fuzzing: a fuzzer may overlook program behaviors (and in turn bug discovery opportunities) if the encoding scheme for map updates treats two distinct coverage facts as if they were the same [16].

For instance, the popular AFL fuzzer [25] tracks edge coverage by combining, upon entering a basic block, the index of the current block with the one of its predecessors as $curr \oplus (prev >> 1)$. Despite a limited run-time overhead, this hashing scheme incurs frequent collisions [16]. Fuzzers such as AFL++ and LIBFUZZER mitigate this problem by inserting dummy basic blocks to disambiguate critical edges [28] in the control-flow graph. Thanks to this transformation, they can track the original edges by using only the (unique) identifier of the currently executing basic block in the modified program, therefore achieving collision-free edge coverage.

B. Points-to Analysis

A points-to analysis is a static program analysis that is able to identify the possible targets of a pointer expression [29] by building the *points-to set* of abstract objects that each expression may reference. An *abstract object* represents an allocation site and concisely captures all the concrete object instances that the program may create there.

Points-to sets are sound, meaning they never miss feasible objects. Sensitivity properties of a specific analysis influence the accuracy of the sets it produces (for the presence of unfeasible abstract objects) and its ability to scale with program complexity. Points-to analyses are nowadays used in several security scenarios (e.g., [30], [31], [32]), also thanks to recent technical advances and state-of-the-art implementations (e.g., [33], [34]) available for mainstream compilers.

In this paper, we use a state-of-the-art points-to analysis to study data-flow diversity properties for function call arguments.

III. MOTIVATION AND OPEN PROBLEMS

We use the code in Listing 1 as a running example to showcase how context-sensitive coverage information can help a fuzzer explore and eventually exploit a faulty program statement that may trigger a bug only when execution reaches it along certain program paths.

The program processes input data as a stream of bytes. Segments of type A1 and A2 contain a variable-size payload of 128 to 192 bytes. Payloads for segments of type B can host up to 127 bytes. For all segments, the payload hosts 16 elements stored adjacently. Element sizes are encoded in the input as 16 consecutive bytes prepended to the payload: these will eventually populate the `sizes` array of the `segment` structure of the program. Accepted inputs contain one segment of type A1 or A2 followed by one segment of type B; the logic enacting this constraint is not shown in the listing for brevity.

```
1  #define MAX_SEG_SIZE  192
2  #define SEG_A12_SIZE  192
3  #define SEG_B_SIZE    127
4  #define EOSEGM(x)     ((x) == 0x23)
5
6  struct {
7    u16 type, len;
8    u8 sizes[16];
9    u8 data[];
10 } segment;
11
12 segment* cur;
13
14 void parse_seg(char* stream, segment* d) {
15   int n = 0;
16   u8 tmp[MAX_SEG_SIZE];
17   for (int i=0; i<16; ++i) {
18     d->sizes[i] = *stream++;
19     n += d->sizes[i];
20   }
21   if (n > MAX_SEG_SIZE) error("too long");
22   for (int i=0; i < n; ++i)
23     tmp[i] = decode_byte(*stream++, d->type);
24   if (!EOSEGM(tmp[n-1])) error("invalid data");
25   memcpy(d->data, tmp, n);
26   d->len = n;
27 }
28
29 void get_seg_A1_A2(char* stream, u16 type) {
30   cur = malloc(sizeof(segment) + SEG_A12_SIZE);
31   cur->type = type;
32   parse_seg(stream, cur);
33 }
34
35 void get_seg_B(char* stream) {
36   cur = malloc(sizeof(segment) + SEG_B_SIZE);
37   cur->type = SEG_TYPE_B;
38   parse_seg(stream, cur);
39 }
40
41 void process_segment(char* stream) {
42   u16 type = decode_type(stream);
43   switch(type) {
44     case SEG_TYPE_A1:
45     case SEG_TYPE_A2:
46       get_seg_A1_A2(stream+2, type); break;
47     case SEG_TYPE_B:
48       get_seg_B(stream+2); break;
49   }
50   // [...] parsing logic continues
51 }
```

Listing 1. Motivating example for context-sensitive fuzzing.

Function `parse_seg` contains a heap-overflow bug at line 25. To trigger it, the program state must satisfy two conditions: (i) the input contains a segment of type B with a stated payload size higher than 127 bytes and (ii) the last payload byte, once decoded, corresponds to the segment termination marker.

In the early stages of fuzzing, a CGF system will have to generate an input containing a segment of type A1 or A2 through progressive mutations of intermediate testcases. This implies that overly long inputs will be rejected at line 16 and that the segment termination marker should appear as the last decoded symbol in the `tmp` buffer to overcome the check at line 24. Both checks lead to immediate program termination.

Later on, once mutations materialize also a segment of type B in the input, a CGF system based on edge coverage may easily change the 16 bytes related to `sizes` to have overly

long payloads meeting condition (i), but will not retain such a testcase for further mutations because its execution does not cover any new edge (or hit count bucket) unless `get_seg_B` is being called for the very first time in the campaign. Therefore, the fuzzer can expose the bug only if condition (ii) is already met by chance when generating such a testcase.

ANGORA [1] extends edge coverage to distinguish executions of the same branch by different calling contexts (defined in Section I). To this end, it dynamically tracks the calling context as the hash of the current call stack, computed by XOR-ing at each call and return instruction the current hash value with the unique numeric identifier of the involved function. Then, it combines this hash with AFL's edge hash identifiers, obtaining a feedback where each map entry should ideally capture a distinct context-sensitive edge instance. We call such kind of feedback **best-effort**.

Challenges: We studied the internal fuzzer wastage that comes with best-effort context-sensitivity approaches by analyzing popular programs from fuzzing literature. We consider two standard configurations of the popular AFL++ fuzzer:

1) EDGES, the context-insensitive AFL-style setup with a coverage map of a standard size of 2^{16} entries indexed by edge hashes (Section II-A);
2) LTO, the configuration of AFL++ optimized for collision-free edge coverage, with unique edge identifiers assigned during link-time optimization. We remark that LTO is currently the most performant setting in the CGF practice.

For context-sensitive fuzzing (CONTEXT), we consider the specific configuration of AFL++ for it (used also in, e.g., [15]), which reproduces the working of ANGORA [1] by combining AFL's edge encoding with the XOR-based call-stack hash described above. We test it in two flavors, using coverage maps of 2^{16} (AFL's default) and 2^{20} (as in ANGORA) entries.

Figure 1 plots statistics collected from a 24-h fuzzing campaign on a subject, `libxml2`, that is particularly representative of the issues behind current approaches. To conduct the experiment, we use the driver and seeds from FuzzBench commit `81d0ed8` and the default timeout of AFL++. We study the *size of the queue*, the *throughput* (completed executions), the number of *distinct map entries covered* by the testcases, and, where applicable, how many per-entry unique *collisions* we identified. A collision at a map entry implies that the fuzzer met and erroneously treated at least two distinct context-sensitive edge instances as if they were the same.

The resulting data highlight two efficiency issues leading to internal wastage for current context-sensitive fuzzers: we will refer to them as *coverage map explosion* and *queue explosion*.

To understand **coverage map explosion** issues, we took a closer look at ANGORA. As acknowledged by the authors [1], their encoding method for context-sensitive edge instances is prone to hash collisions: we identified them on **50.7%** of the map entries for the CONTEXT 2^{16} fuzzer configuration.

Collisions are undesirable, since they lead to loss of context-sensitivity[1] and ultimately increase the likelihood of discarding useful testcases [16]. Therefore, ANGORA uses a larger map with 2^{20} entries. While this choice can effectively mitigate collisions (1.2% for CONTEXT 2^{20}), it can hamper the

Fuzzer configuration	Queue size	Executions / sec (large L2)	Map entries Used / Total	Colliding
EDGES (2^{16} map)	9 911	609.04	19.86% of 64 KB	9.8%
LTO (collision-free)	11 093	572.02	15.59% of 50 KB	-
CONTEXT (2^{16} map)	**33 675**	530.10	79.54% of 64 KB	**50.7%**
CONTEXT (2^{20} map)	21 157	**84.38**	7.21% of 1 MB	1.2%
PREDICTIVE	15 455	490.62	9.28% of 256 KB	-

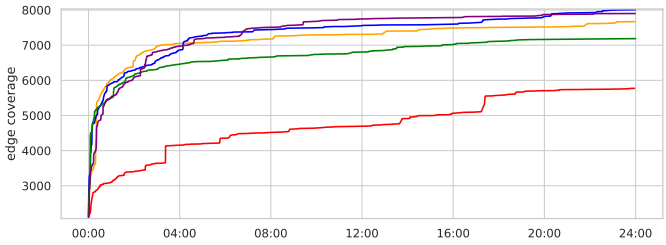

Fig. 1. Fuzzer's internal wastage vs. edge coverage over 24 hours with best-effort context-sensitivity. Peak values for degradation are highlighted in bold.

throughput of the fuzzer because of higher map access latency (as the map would no longer fit common L2 cache sizes, which can accommodate up to 2^{18} entries) and slower processing at the end of each execution. On standard hardware, we observed induced slowdowns of one order of magnitude.

To partially mitigate this coverage map explosion problem, we collected our data on a high-end Intel Xeon Platinum 8160 with a 1-MB L2 cache. Even on such a high-end configuration, the number of completed executions dropped from ~45 millions to ~7 millions. Such low fuzzing throughput ultimately resulted in much poorer (context-insensitive) edge coverage after 24 hours than any other configuration.

The second problem, **queue explosion**, is well-understood in literature: as observed in [10], while retaining more seeds offers "stepping stones for more meaningful mutations that lead to final crashes, [retaining] too many of them would hurt the fuzzing performance" as the differences between most such seeds are likely so tiny that would hardly result in new bugs.

For the CONTEXT 2^{16} configuration, the queue size grows significantly (from 9 911 to 33 675 retained testcases), but the edge coverage achieved over time is appreciably lower than EDGES (where 9.8% of map entries see collisions) and much lower than the one obtainable with a collision-free LTO solution. The problem is less noticeable in the CONTEXT 2^{20} configuration (although the queue size still doubles to 21 157), but only because the much lower throughput (and edge coverage) masks the queue explosion problem.

Summarizing, our analysis shows that current context-sensitive strategies (CONTEXT) struggle to achieve good precision without introducing internal wastage due to explosion issues: by allowing more collisions, they lose context-sensitivity (at the cost of discarding important testcases), whereas by reducing collisions, they overly discriminate contexts (at the cost of retaining too many testcases and trashing the fuzzing

[1]And, even worse, weaker path sensitivity than a context-insensitive baseline, since a single hash is used for calling contexts and edges. Therefore, one may suggest combining a collision-free edge ID with a hash of the context. Unfortunately, this method would be much poorer than the one of ANGORA due to the limited entropy of edge identifiers, which would be completely marginal compared to the one of contexts.

throughput). The performance of CONTEXT falls behind by an appreciable margin not only the collision-free edge coverage setting of LTO, but even EDGES. Best-effort context-sensitivity was similarly outclassed for bug finding capabilities in the full evaluation that we will illustrate in Section VI-A (Table III).

The key reason why this is essentially an impossible needle to thread is that prior strategies are entirely *blind* to which of the many distinct contexts are important to capture in order to retain *interesting* testcases. As an example, `libxml2` can see potentially up to 16-million distinct contexts originating from its `main`; more in general, their number is often exponentially large w.r.t. the number of program functions [11].

Our Approach: In this paper, we explore a *selective* angle to deploy context-sensitive fuzzing in a more effective way: we augment only certain program regions with contextual information, devising then a novel *predictive* solution to statically identify regions that are likely to benefit from context-sensitive profiles for the edges traversed during execution.

As a concrete instance of this strategy, we favor call sites that see a higher diversity for the incoming data-flow at call arguments. For our example, such a predictor would recognize that the `segment` object flowing into the buggy function comes from different allocation sites depending on the caller.

Then, as we study only data-flows for function arguments across different call sites, instead of the full calling-context we can rely on a much lighter context abstraction that discriminates only the identity of the caller function.

Our approach (PREDICTIVE) augments an LTO-style map with entries for collision-free context-sensitive profiles of edges from selected regions. For the rest of the code, we use collision-free context-insensitive edge tracking as LTO does. We bound our selection so that the map fits standard L2 caches.

Ultimately, all these choices allow us to hit the "sweet spot" between insufficient and excessive context-sensitivity, uncovering more bugs in well-known benchmarks with only a moderate impact on the fuzzer's internal wastage.

IV. PREDICTIVE CONTEXT-SENSITIVITY

This section presents the three main pillars of our approach:

1) a collision-free method to encode context-sensitivity;
2) a selective approach to restrict context-sensitive fuzzing to program regions of interest for the sake of scalability;
3) a data-flow analysis to predict regions likely to benefit from having been selected when a coverage-guided exploration reaches them.

We produce a transformed program containing context-sensitive instances of control-flow edges, added according to a user-specified budget and in a cost-effective manner. Existing CGF systems can test it without requiring any changes.

A. Function Cloning

A way to turn a context-insensitive program analysis into a context-sensitive one is to expose to the analysis a separate instance (*clone*) of the code unit of interest at each different encountered context. For instance, if contextual information is represented only by the caller of a function, the analysis may produce separate results for the unique clones of the callee devised for each possible caller.

Such an approach has two main advantages: it offers backward compatibility for existing fuzzing instrumentation solutions and can accommodate different context-sensitivity definitions. Let us consider calling-context information, initially on recursion-free programs for simplicity.

One may disambiguate the calling context for a specific function by taking the call graph of the program and, for each maximal acyclic path that reaches the function, introducing a clone at every caller-callee pair on the backward walk to its root node. In this way, whenever the analysis reaches a clone of the original function, the path from the root function to it is unique. Therefore, the identity of the clone is sufficient to precisely determine the invocation context.

To handle recursion, we look for functions involved in direct and indirect recursion by analyzing the strongly connected components (SCCs) of the call graph [35]. During path analysis and cloning, we treat each SCC as a single node without a self-edge. This allows us to retain precise contextual information before and after entering recursive sequences (which in general may be unbounded in depth), treating only the recursive parts in a context-insensitive manner.

For a coverage-guided fuzzer, we need a way to discriminate different clones of a function of interest that is both cheap to maintain or retrieve at run-time and composable with other encoding techniques in a space-efficient and collision-free way.

An elegant and effective way to maintain context-sensitivity for program points is to manipulate the code of the program and add concrete copies of the involved functions. This choice brings several advantages. By exposing contextual information through new code locations, we offload the collision problem to the feedback mechanism already in use by the coverage-guided fuzzer. With edge coverage, existing collision-free edge encodings will just assign **unique (context-sensitive) edge identifiers** to code from clones. Therefore, function cloning effectively solves the collision problem we saw in Section III.

Furthermore, when deploying context-sensitivity in the *selective* flavor that we present in the next section, another advantage of our scheme is that it brings virtually no run-time overhead for tracking and retrieving the context, as we trade this efficiency for a modest increase in program size.

Let us use as running example our program from Listing 1. The relevant caller-callee pairs are (`get_seg_A1_A2`, `parse_seg`) and (`get_seg_B`, `parse_seg`). For simplicity, we pick the second for specialization as we know that such path can expose the bug at line 25. Our cloning primitive adds to the program a duplicate of `parse_seg`, which we call `__clone_ps`, and patches the call at line 38 to invoke it in lieu of the original function. When a coverage-guided fuzzer executes the augmented program, the branch originally at line 22 will benefit from separate coverage information when reached via `get_seg_B`, allowing the fuzzer to treat it as an interesting testcase (and, in more detail, to become sensitive to the different payload lengths that its hit count may capture).

By choosing to work on call sites, we can virtually model any notion of context-sensitivity based on tracking portions of

TABLE I. CODE FEATURES OF FUZZBENCH SUBJECTS.

Benchmark	Type	Edges	Functions	Call sites	Calling contexts
ffmpeg	C, some C++	716 046	5 314	44 500	8 014 021
file	C, some C++	15 986	250	985	19 217
grok	C++	94 092	535	2 234	11 025
libarchive	C	67 096	866	4 377	27 984 301
libgit2	C	107 785	1 718	5 467	3 024 953
libhevc	C	119 646	197	853	125 907
libhtp	C++	11 203	181	706	6 718
libxml2	C	104 351	1 147	6 708	44 652 617 060
matio	C	24 112	300	1 795	2 793 663
muparser	C++	14 007	103	483	6 120
ndpi	C	49 216	355	1 991	10 507
njs	C	57 402	588	3 818	12 671 908
openh264	C++	78 819	384	1638	28 441
stb	C/C++	11 861	144	881	11 501
usrsctp	C	96 225	405	4 303	3 294 931 527
zstd	C/C++	38 863	848	5 027	140 141

Algorithm 1: Priority-based Cloning

```
function CloneByPriority(program, budget)
    callsites ← ⋃_{f∈program} GetAllCallsites(f)
    priorities ← GetPriorities(callsites)
    pqueue ← PriorityQueue(callsites, priorities)
    while program.size < budget do
        callsite ← pqueue.pop()
        target ← GetCallTarget(callsite)
        new_target ← CloneFunction(target)
        SetCallTarget(callsite, new_target)
        new_callsites ← GetAllCallsites(new_target)
        new_priorities ← GetPriorities(new_callsites)
        pqueue.push_all(new_callsites, new_priorities)
```

the call stack: a global policy will ensure that each cloning action draws out a piece of the desired portion. The call sites present within an added clone may be in turn disambiguated for context-sensitivity by applying cloning recursively.

B. The Need for Selective Sensitivity

While cloning can expose context-sensitivity information for program points in a "fuzzer-friendly" manner, it does not help us get around the path explosion problem that comes with calling contexts (Section III). As evidence of this issue, Table I reports statistics collected for programs from the FuzzBench test suite that we later use for evaluation purposes (Section VI).

As a fuzzing harness often tests only a relevant subset of a code base, we collect the figures after removing all the functions unreachable according to LLVM's static analyses. In the *edges* column, we report the number of basic blocks that a collision-free edge coverage scheme instruments after breaking all the critical edges in program functions [26]. The last three columns represent, respectively, the number of nodes, edges, and acyclic paths in the call graph.

For many subjects, the number of contexts appears intractable for any practical collision-free attempt (we will return to this in Section VII), including cloning. Even when the contexts are not millions or more, the number of "context-sensitive" edges to disambiguate may still increase dramatically when the call sites are many, requiring in turn (inefficient) large coverage maps for their (collision-free) tracking.

However, we argue that a much more effective approach is possible: adding context-sensitivity only to *selected* program portions. Algorithm 1 presents the high-level workflow: we process the call graph at call-site granularity and follow a *prioritization policy* to pick individual call sites for cloning. As a baseline, we consider a *random* policy that prioritizes them uniformly at random.

We surveyed static analysis literature for contextual information representation in the programming language community (e.g., [36], [37], [38]) and derived three policies that approximate their core ideas by performing a visit of the call graph and assigning priorities (captured by visit order) according to topological properties:

- *top*: assigns higher priority to call sites from nodes closer to the root(s) of the call graph, progressively exposing the context in a top-down fashion as in [37].
- *bottom*: assigns higher priority to call sites closer to leaves. This policy progressively exposes the last entries on the call stack as in *call strings* [36], which in some domains can effectively replace the full calling context.
- *uniform*: treats every call site with the same priority. It resembles [38] and mixes the effects of the other policies, exposing the top or bottom call-stack entries leading to a node depending on its proximity to a root node or a leaf.

In preliminary tests[2], these policies exposed a few more bugs than standard edge coverage (thus already outclassing best-effort context-sensitive solutions) and did not experience any evident internal wastage. However, their apparent benefits were modest and also difficult to understand when compared to *random*, as the policies often resulted in similar performance.

Eventually, we looked at these results retrospectively. Policies of this kind are well suited for static program analysis scenarios, where partial contextual information may still expose to an analysis sufficient information to reason on all the possible *refined* program states and, in turn, the user can measure the improvement (if any) in the precision of the returned answers. Instead, coverage-guided fuzzing is a dynamic analysis technique based on a lightweight abstraction of program state: no direct static measurement of the benefits of context-sensitivity seems possible. To effectively take advantage of any added context-sensitivity (which can be available only in a limited quantity), we concluded that we need a predictor for program portions that may practically benefit from it during fuzzing.

C. Data Flow-based Prediction

A pivotal element of our proposal is a *prediction*-based policy that prioritizes for cloning those call sites where the callee sees higher **diversity** in the incoming data-flow compared to other uses of the same function in the rest of the program. Specifically, we favor cases where the abstract objects potentially incoming as arguments for the callee function are more peculiar (*i.e.*, less frequently met) w.r.t. other call sites where the function is invoked. Our hypothesis is that such diversity can be a promising indicator that the program may enter "less common" internal states along these execution contexts.

Prioritizing such contexts for cloning and, in turn, retaining testcases that hit them during execution may allow the fuzzer

[2]The results for top (shown as 'bfs') and uniform can be found at https://www.fuzzbench.com/reports/experimental/2021-05-25-cloning/index.html, whereas for random and bottom at https://www.fuzzbench.com/reports/experimental/2021-07-09-cloning/index.html.

to delve more pervasively into these behaviors, both locally at the callee and in any subsequently reached code that is affected by the data flow. As we will explore in Section VI, the analysis we present below turns out to be a good predictor in practice for eliciting profitable states and uncovering new bugs.

We argue that function arguments are a natural way for programs to orchestrate data-flows through their code units. Therefore, we study the invocation of every function at its different call sites in the call graph and analyze what values are possible for each of its arguments. We prioritize cloning those call sites that pass as arguments *abstract objects that never or rarely appear at other call sites*.

In other words, we find it reasonable to differentiate those call sites (*i.e.*, to introduce clones for callees) that see peculiar incoming objects, while we predict a lower benefit from doing so at call sites that see objects that recur at other places too. For example, for a function with two call sites, we have little interest in cloning it if the two pass similar objects; instead, when the two pass very different objects, we find it reasonable to differentiate them for the fuzzer to explore both.

In this paper, we focus on pointer-type arguments and use an off-the-shelf analysis to build points-to sets (Section II-B), obtaining the possible abstract objects that an argument may reference when passed at a call site. We compute the prediction to use as priority value in Algorithm 1 as follows. Let the target function be in use at n call sites in the call graph[3] and O be the set of all abstract objects that may be passed via its arguments at the current call site. The priority p of the call site is:

$$p = \frac{1}{n} \times \sum_{o \in O} (n - n_o)$$

where n_o is the number of call sites for f where object o may appear in any of its arguments. As we said earlier, we seek to favor the diversity of the incoming data-flow: an object o that does not appear at other call sites for the target will contribute with a $n - 1$ addend, whereas an object that may appear at all call sites will give a zero addend. Eventually, the edge coverage collected for the clones exposes the incoming data-flow diversity to the fuzzer, favoring a more pervasive exploration of the underlying program states.

D. Discussion

With our approach, we propose to overcome the precision and efficiency limitations of current context-sensitive fuzzing flavors by augmenting only selected program points with contextual information. Our data flow-driven prioritization policy shows promise in practice, retaining for further mutations inputs that eventually led us to discover new (or more) bugs.

In our approach, we chose to focus on pointers because pointer diversity always leads to data-flow deviations, while non-pointer diversity does not necessarily do so. We also believe memory errors to be more likely in presence of data-flow deviations, and fuzzers are notoriously effective in exposing them [39] (especially in combination with sanitizers [40]). An interesting follow-up may be to study what non-pointer variables in a program can lead to "helpful" diversity and, in turn, to what extent. In this scenario, a practical aspect to account for is the precision of value analysis techniques for non-pointers (e.g., value range analysis [41] on integer arguments), as too coarse results could mask real diversity.

Compiler-based instrumentation is a natural way to deploy our approach. For fuzzing programs available only as binaries, binary rewriting techniques or a modified runtime can intercept and divert call sites. However, analyzing pointer arguments may be challenging as, among others, it would need to recover object locations. We leave this investigation to future work.

V. Implementation

We implement our techniques as a set of analysis and transformation passes (~2k C++ LOC) for the intermediate representation (IR) of the LLVM compiler, a popular choice for fuzzers that instrument source code. We operate on a link time-ready whole-program IR file that the GLLVM helper [42] obtains for the uninstrumented program. We produce a transformed IR file and feed an off-the-shelf fuzzer with it.

As for evaluation purposes we opted for the state-of-the-art AFL++ [14] fuzzer (version 3.15a), we devise a simple Python helper that automates the compilation process and also the insertion of sanitization machinery. Our cloning pass has provisions to correctly handle the instrumentation introduced by popular sanitizers such as ASAN and UBSAN, which insert tripwires that help fuzzers expose silent bugs [43].

For sizing purposes, we implement an analysis to estimate, for each cloning decision, the coverage map size increase due to the unique identifiers that the collision-free edge coverage encoding of AFL++ would introduce for the clone. We simulate a cloning action and reuse AFL++'s instrumentation algorithm to count the edge entries the clone would need in the map.

Good fuzzing practices [1] recommend map sizes no larger than standard L2 cache sizes (i.e., 256 KB), whereas overly large maps can be detrimental for performance even on favorable hardware, as we saw in Section III. Once we set a maximum desirable map size, we can use as residual budget for cloning the "free" map entries after we accounted for the edges currently in the program and, potentially, add clones up to its exhaustion. Our evaluation sets a budget of 256 KB, which can host up to 2^{18} map entries. In practice, this tuning choice allowed our fuzzers to discriminate and pervasively delve into new program states without incurring internal wastage.

To analyze pointer arguments at call sites, we use the state-of-the-art points-to analysis `FlowSensitive` from the popular SVF framework [33]. Among the analyses implemented in SVF, it is expected to bring the most accurate points-to sets for general code, as it carries an Andersen-style analysis enhanced with field- and flow-sensitivity (while it remains array- and context-insensitive for the sake of scalability).

As an implementation refinement, we attempt to lower the priority of a recurrent class of uninteresting call-site targets: error-handling functions that lead to program termination. In the programs we study, many such functions see a very high number of callers and, consequently, an inherently diverse incoming data-flow at various call sites. We opt for lowering the priority of the call sites whose target is a function called by at least 25% (a value set empirically) of all functions in

[3] We remark that we compute priority values on the unmodified program.

the program. We have verified that this choice affected only error-handling functions in our tests.

Our prototype can also attempt to reason on paths involving indirect-call sites, by promoting each indirect call into a conditional selection of direct calls to plausible targets [44], [45], [46]. However, this is disabled by default since precise reasoning on indirect calls is notoriously hard. With a static approach, the precision of the analysis for building call-target sets is crucial [47]: in most of the cases we analyzed using points-to analysis, the size of the resulting sets led to path explosion. Nonetheless, as we will see throughout Section VI, the effects of our techniques allowed us to expose bugs and report security vulnerabilities in heterogeneous programs written in C/C++ and object oriented-style C. As future work, we plan to explore the potential benefits of profile-guided indirect call promotion [44] for these subjects, for instance using testcases from a short fuzzing session, as well as of recent advances in static type-based dependence analysis techniques [48].

VI. Evaluation

We study the performance of predictive context-sensitive fuzzing using the FuzzBench testing infrastructure. Popular in academia and industry since its release in 2020, FuzzBench has become a de-facto standard benchmarking platform and program collection for fuzzing research. FuzzBench targets real-world programs, pinning specific versions for reproducibility and result validation [17]. We select the `'type: bug'` configuration of FuzzBench, a choice made also in other recent bug-oriented studies [49], [50], [51]. We study different dimensions of our approach for the following research questions:

RQ1: Can we outperform the state of the art in bug finding? Can we find vulnerabilities that existing approaches overlook?
RQ2: To what extent do we induce internal wastage, if any?
RQ3: What burden do we place on the compilation pipeline?

Atop the AFL++ [14] fuzzer, we test these configurations:

- **`context`**: *best-effort* context-sensitivity as evaluation baseline, using the implementation available in AFL++ that reproduces what proposed in ANGORA [1];
- **`lto`**: collision-free edge coverage boosted with link-time optimization. It is the the most effective setting available for context-insensitive coverage-guided fuzzers [14] and serves as a reference point to show (in further detail than in Section III) the internal wastage effects of `context`;
- **`predictive`**: the approach we propose in this paper;
- **`random`**: an uninformed prioritization policy serving as a baseline for selective context-sensitivity;

For `context`, we use a coverage map of 2^{18} entries to fill the L2 cache (256 KB) typical of most machines, including the FuzzBench cloud infrastructure on which we ran our tests. We do not evaluate larger sizes as we experienced significant internal wastage for the reasons discussed in Section III. We also remark that `context` reproduces only ANGORA's context-sensitive edge coverage encoding: that is, it does not perform the taint tracking or gradient-descent based search that are other distinctive features of ANGORA. The reason for it is that we want to stress context-sensitivity alone (which other fuzzers, like WEIZZ [15], already use): the independent contributions of such features would only pollute the analysis.

For `lto`, the number of instrumented edges in each program (Table I) determines the map size.

For `predictive` and `random`, we use the largest cloning budget value such that the resulting map still fits[4] an L2 cache of 256 KB (i.e., up to 2^{18} entries).

We could obtain a compilable whole-program IR file (Section V) for 16 of the 22 benchmarks from FuzzBench. Bugs and missing features in the GLLVM [42] helper[5] and other compilation errors unrelated to our techniques prevented us from testing the other programs. The link-time primitives that recently became available in LLVM may help for them for future implementation extensions.

For all the fuzzer configurations that we study, we instrument each whole-program IR file with the ASAN and UBSAN sanitizers [52] to expose common classes of silent bugs. All the fuzzer configurations that we test work on binaries built with `-O3` optimization level.

A. RQ1: Effectiveness in Bug Finding

To evaluate the bug finding capabilities of our four fuzzer configurations (hereafter *fuzzers* for brevity), we initially rely on the infrastructure of FuzzBench to count unique bugs via automatic crash deduplication based on unique stack traces. As we run the fuzzers on its cloud platform, each configuration-benchmark pair sees 20 trials of 23 hours each.

1) General Trends: Following standard practices [53], we reason on the median values over all trials to mitigate the well-known effects of randomness in fuzzing. Figure 2 reports the boxplots for each benchmark showing the number of bugs found by each fuzzer. For each benchmark, the fuzzers appear in the ranking order given by their median number of bugs found across the trials and using their maximum number to break ties when necessary.

To compare the effectiveness of each fuzzer, we first consider the *average score* metric from FuzzBench. For each benchmark, the score of a fuzzer in a `'type: bug'` campaign is given by expressing the median number of bugs[6] it finds as the percentage of the median number of bugs from the fuzzer that performed best on that benchmark. The final cross-benchmark average score for a fuzzer, shown in Table II, is the average of individual benchmark scores and mitigates distortion effects due to benchmarks having a different number of total bugs [17]. We note that cross-benchmark average scores reflect the relative performance of each fuzzer in one experiment setting: therefore, they do not generalize for comparisons with other selections of fuzzers and/or programs.

The best-performing fuzzer is the one using our predictive policy: `predictive` obtains the highest score with an 11.84 net difference with `lto`, which in turn largely outperforms

[4]Except for `ffmpeg`, for which the number of unique edges requires more than 2^{18} entries already with `lto`: therefore, we set the budget for it to the nearest feasible multiple of two (768 KB).

[5]Two practical limitations we observed with GLLVM are i) its incorrect handling of source files that a build system may supply to a linker (while this may seem an unorthodox behavior, both clang and gcc allow it; we reported the issue to its developers) and ii) when it invokes `llvm-link` to merge the bitcode files, the IR elements for indirect functions (GNU IFUNC) are lost.

[6]Coverage-centric experiments use the median code coverage instead.

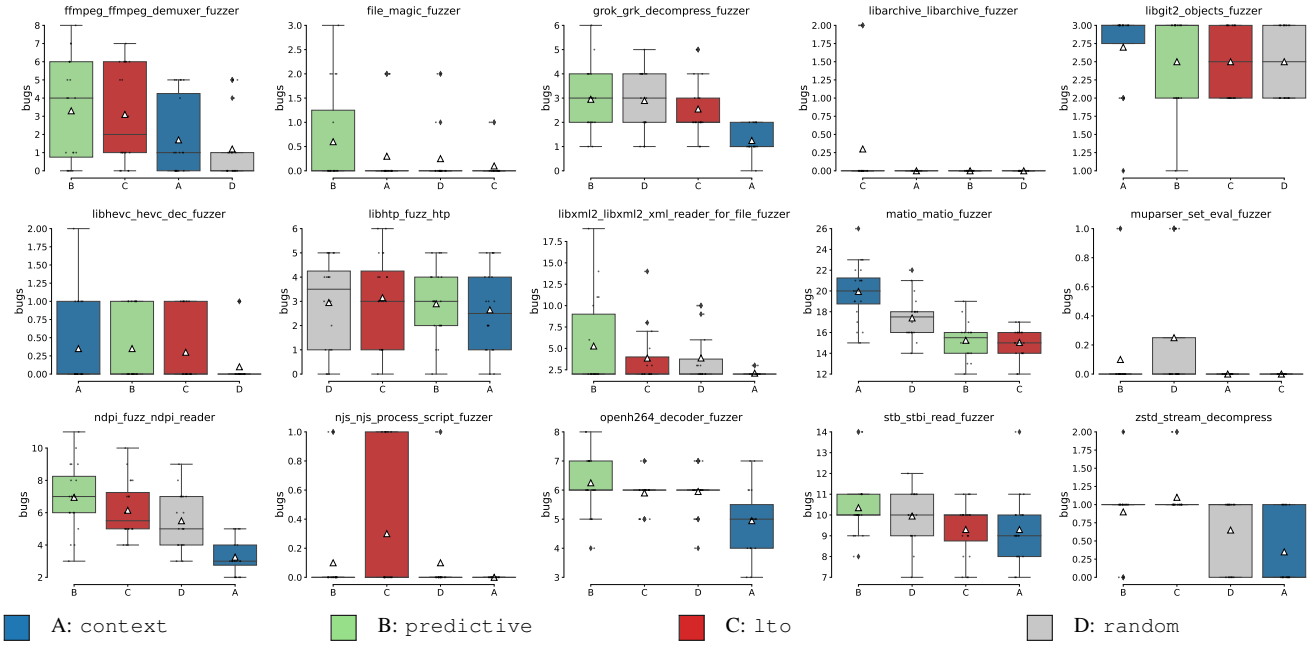

Fig. 2. Boxplots with mean value (△) and raw data points (·) for bugs uncovered in the FuzzBench programs across 20 trials. Fuzzers are ordered by ⟨median, maximum⟩ number of bugs found. We leave out `usrsctp` as no fuzzer found bugs for it.

TABLE II. CROSS-BENCHMARK AVERAGE SCORE FROM FUZZBENCH.

Fuzzer configuration	FuzzBench score
`predictive`	94.14
`random`	82.98
`lto`	82.30
`context`	63.42

`context` (and even `random` does too). The `predictive` fuzzer will similarly stand out also in the analysis of individual bugs that we provide in the next section.

As we move to the other fuzzers, we remark how the `lto` state-of-the-art configuration is a *strong baseline*. In addition to collision-free encoding of edges, which outperforms classic (collision-prone) edge tracking and refinements [16], it benefits from link-time optimizations such as additional inlining. For instance, LLVM may inline a short-sized callee at a call site for performance, incidentally providing *some* context-sensitivity [54] as the inlined edge instances get new identifiers. However, an optimizing compiler follows performance-based (rather than context sensitivity-based) inlining policies. When our data flow-based prediction mechanism drives the cloning decisions, we can observe a significantly larger number of bugs found for the subjects considered in this evaluation.

On the contrary, the best-effort context-sensitivity of `context` suffers from a combination of the problems analyzed in Section III. While we defer a detailed discussion of internal wastage effects to Section VI-B, collisions hamper its ability to distinguish, and thus explore, useful program states that not only `predictive`, but even `lto` can often retain in its queue. Combined with the time spent analyzing likely uninteresting testcases that pollute its queue and the lower end-to-end throughput (Section VI-B), `context` ranks on average as the least effective fuzzer configuration in our tests.

2) In-depth Analysis: We now qualitatively analyze the unique bugs identified by the fuzzers `predictive` (125), `lto` (112), and `context` (102). We leave out `random` (110) for brevity. We start by discussing the left part of Figure 3, which compares the unique bugs found by `predictive` against the `lto` and `context` fuzzers, which embody the state of the art in context-insensitive and sensitive fuzzing. Table III lists how many bugs we found on each subject.

Due to internal wastage effects, `context` **missed** 27 of the unique bugs that both `predictive` and `lto` could find. Of the 102 unique bugs `context` found, 74 were found by both the others, and 82 by `predictive`. As for the 18 bugs found only by `context`, 15 are from `matio`—on which, as we discuss next, our predictive strategies are less effective.

On the other hand, `predictive` revealed twice as many (43) unique bugs missed by `context`, found in 9 of the 16 subjects we study, and 23 more bugs in total (**+22.55%**). Finally, the two fuzzers find an identical number of bugs in 5 subjects. We conclude that our approach significantly outperforms the state of the art in context-sensitive fuzzing.

Comparing the counts for `predictive` and `lto`, the former found 13 more bugs in total (**+11.6%**). Also, 24 of its 125 bugs (**19.2%** of the total) were missed by `lto`; this amount equals the 21.4% of the `lto` count. Of the 112 bugs found by `lto`, our approach missed 11 bugs (10.7% of the `lto` count). Hence, our approach not only significantly outperforms best-effort context-sensitivity, but does not show appreciable internal wastage compared to `lto`. With more and different bugs found, we may argue that our approach has benefited the exploitation work of the fuzzer (Section III).

Testcase Dissection: To better understand these results and how refined contextual information may be behind the bugs that only `predictive` found, we analyze several char-

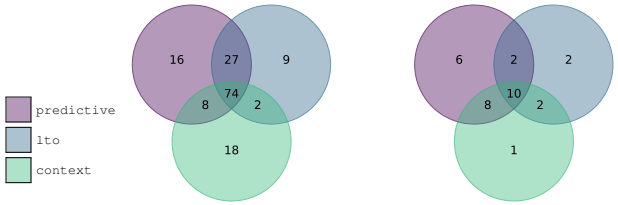

Fig. 3. Venn diagrams for unique (left) and also new (right) bugs found by the fuzzers across all benchmarks.

acteristics of the crashing testcases. For example, we examine how often the bugs come from code that the state-of-the-art `lto` fuzzer driven by edge coverage (possibly refined by LTO effects) reaches in its exploration without causing a crash.

Although our methods improves context-sensitive fuzzing, the following discussion mainly considers `lto` as a comparison point both due to its significantly better bug finding performance compared to `context` and for understanding why context-sensitivity helped our fuzzer's performance.

We downloaded the queues from all trials from the FuzzBench cloud and ran each testcase on a locally compiled binary. We successfully reproduced 23 out of 24 bugs; the one failure involves subject `grok` and a corrupted zip file stored on the FuzzBench cloud for the trial that exposed the bug.

We check the bugs found only by `predictive` against the *cumulative code coverage* achieved by `lto` on each of its 20 runs. We note that 16 bugs occur in code that `lto` covered in at least one trial without yielding a crash. Conversely, 7 bugs are from new code coverage.

The attentive reader may find the last result surprising. Typically in program analysis, context-sensitive approaches are meant to improve the results (here, the exploitability) for code that is already within the reach of context-insensitive solutions. However, context-sensitivity may make a fuzzer retain a testcase that, through further mutation, eventually "unlocks" new coverage by meeting particular control-flow conditions (e.g., branch predicates) later in the execution. Such testcases progressively lead `predictive` to buggy parts that `lto` never reached in our tests across 20 trials. In Section VI-B we will provide code coverage statistics for all subjects.

Moving forward in the analysis, we observe that for 14 bugs cloning choices directly contributed to reaching the buggy program location and associated state, as one or more cloned functions were active on the call stack upon the crash. In 21 bugs, contextual information helped by retaining "ancestor" testcases during previous mutations: we measure this property by seeing if the crashing testcase exercises context-sensitive edges from one or more clones (which the fuzzer would see as a novelty). These allowed the fuzzer to further mutate the input and, in turn, the induced program state until exposing the bug. We attribute the remaining 2 bugs (1 of which was covered but not exploited by `lto`) to fuzzing entropy.

Section VI-A3 will complete this discussion with a case study that further showcases how data-flow diversity is a good predictor of regions for which retaining testcases that reach them from different contexts can be beneficial when fuzzing.

We also reviewed the bugs found by both `lto` and `predictive`. An interesting finding was that in 12 cases

TABLE III. UNIQUE BUGS FOUND PER BENCHMARK BY THE FUZZERS.

Benchmark	Language	context	predictive	lto	random
`ffmpeg`	C, some C++	6	**11**	10	7
`file`	C, some C++	**3**	**3**	1	2
`grok`	C++	2	**7**	6	**7**
`libarchive`	C	0	0	**2**	0
`libgit2`	C	3	3	3	3
`libhevc`	C	**2**	1	**2**	1
`libhtp`	C++	5	5	**6**	5
`libxml2`	C	3	**22**	16	12
`matio`	C	43	**26**	**26**	34
`muparser`	C++	0	**1**	0	**1**
`ndpi`	C	12	17	**18**	13
`njs`	C	0	1	1	1
`openh264`	C++	7	8	8	8
`stb`	C/C++	15	**18**	11	15
`usrsctp`	C	0	0	0	0
`zstd`	C/C++	1	**2**	**2**	1
Total	All (16)	102	**125**	112	110
	C only (8)	63	**70**	68	64
	C++ only (4)	14	**21**	20	21
	Mixed (4)	25	**34**	24	25

the two fuzzers exposed the same unique bug from testcases with a different edge coverage, suggesting that `predictive` found the bugs from a different angle.

As for the bugs found by `lto` and missed by `predictive` (11), we attribute them to the different exploration and scheduling choices that the two fuzzers follow. In particular, context-sensitive fuzzing is designed to spend fractions of the fuzzing budget in mutating testcases (and exploring "pervasively" the associated program states [1]) that `lto` does not retain. However, `predictive` retains any testcase that `lto` would: hence, those bugs remain in its reach.

We remark that this trend is expected: in the given time, our approach prioritized and explored other program parts better. Such a trend is common for fuzzer specializations: for example, also Token-level AFL [55] finds diverse and more bugs than prior concepts, missing a few for the same reasons. The ability to find different bugs is a pillar for initiatives like OSS-Fuzz that stack different fuzzers and motivates recent research on ensemble fuzzing [56], [57]. The **inclusion relations** of Figure 3 are meant to show such differences, which may be overlooked if one looks at bug counts only. We refer our readers to Appendix A for detailed per-benchmark statistics.

On a different note, when looking at Table III, the bug finding capabilities of our approach do not appear to reflect a strong influence from the source language.

Instead, a subject worthy of a detailed discussion is `matio`. It is the only subject on which `context` is the best performer (and also by a large margin). Written in C and featuring a high number of potential calling contexts (Table I), `matio` follows an object-oriented paradigm that heavily relies on evolving the state of a single object by manipulating its fields while passing it across many functions. This dynamic is missed by our diversity heuristic, which sees objects as a whole. As a result, we may clone call sites that are much less appealing than those that `context` explored blindly but

profitably; here, random outperforms predictive by 8 bugs. On the contrary, predictive is the most effective fuzzer on libxml2, another target written in C following an object-oriented paradigm and with a huge amount of potential calling contexts. Complex state variations as in programs like matio deserve further investigation, for example combining our approach with the data-oriented feedback of [58] from likely invariants for recurrent program variable values.

3) New Bugs: As a last dimension to investigate, we conduct a set of experiments on the ability of the fuzzers to find lingering bugs in well-tested software. We again leave out the less performant random fuzzer.

We first analyze whether any of the bugs we found would affect the latest program versions at the time of testing (February 2022), which follow those used in FuzzBench by 9 months to over 4 years. These programs are tested daily by the OSS-Fuzz initiative and are recurrent choices for many papers behind recent fuzzer concepts. As the right part of Figure 3 shows, 31 unique testcases (as deduplicated by FuzzBench) could crash recent versions too: in particular, predictive (26 bugs) outperforms context (21), which in turn found only one bug that predictive or even lto did not.

For the 31 testcases, we ruled out a few that matched issues in existing public bug reports and responsibly disclosed all the others to the respective developers. For bugs that hinted at ostensible security issues, we conducted further manual analysis to identify the logical root cause underlying each bug and cluster them accordingly (that is, we "conceptually" merged some). This analysis exposed 8 security issues in 5 programs: ffmpeg (1), njs (1), stb (4), libhevc (1), and matio (1). Six of them received a CVE ID (Appendix A), 1 was deemed a duplicate of one of our newly assigned CVE IDs (stb), and 1 was not considered a vulnerability by the vendor according to its criteria (we reported an undefined behavior from an invalid shift in libhvec). From commit dates, the issues had been present in the programs for at least 1.5-3 years.

In more detail, 5 issues derive from bugs found by predictive only: 3 for stb and 1 each for ffmpeg and libhevc. For these issues, lto typically covered the involved code without triggering a crash, with the exception of 1 issue as it involved new code coverage. The remaining 3 issues came from bugs spotted by both fuzzers[7].

As a case study, we discuss one of the CVEs assigned for stb, which is a well-tested image processing C library. The bug showcases how context-sensitivity is helpful to expose overlooked buggy code and how our predictive mechanism made effective cloning decisions for that end. The bug manifests as a heap use-after-free violation caused by an out-of-bound array write during JPEG decoding. The vulnerable function stbi__process_marker does JPEG segment processing and is invoked, in a strict order, from two call sites: in the initial header parsing of stbi__decode_jpeg_header and in the subsequent image decoding of stbi__decode_jpeg_image.

The incriminated code derives from the input a quantity, hereafter n, that controls the trip count of two loops that populate two arrays of 256 bytes within a complex structure. In the crashing testcase generated by predictive, the out-of-bound accesses impact, among others, data that another function, stbi__jpeg_huff_decode, later uses to index memory. This results in a negative offset that makes the rogue pointer reference memory previously allocated by the fuzzing harness for another testcase. Upon the crash, stbi__decode_jpeg_image is still present in the stack trace for stbi__jpeg_huff_decode.

Context-insensitive fuzzers cover the edges of the vulnerable function, but do not differentiate the program internal states when invalid segments reach it from the second call site because they see no new coverage. In more detail, hit count buckets are of limited help because of the counts seen when processing header segments. Also, after invoking the buggy function, stbi__decode_jpeg_header carries validity checks on several input bytes following the segment. Invalid n values induce "unaligned" reads, and the validity checks discard the input when such bytes do not meet the expected semantics. As a result, the fuzzer has limited wiggle room to retain and further mutate testcases leading to increasingly higher values for n for invocations from the second call site.

Our approach introduces context-sensitive instances of the loop edges: the fuzzer becomes sensitive to different n values induced from the second call site, so it will retain and further mutate the associated testcases, eventually exposing the bug. In our tests, we measured that our predictor selected the call site for cloning with a priority $p = 0.91$ (we recall that $p \in [0, 1]$).

B. RQ2: Internal Wastage

As seen in Section III, internal wastage may hamper the effectiveness of a fuzzer by making it explore uninteresting program states and/or face higher latencies for completing the execution cycle of each testcase. We studied its impact by collecting in Table IV statistics on the queue size and the execution throughput of each fuzzer at the end of the fuzzing session.

Our predictive fuzzer sees a median queue size per benchmark that, on average, is moderately larger (**+26.4%**) than the value measured for lto. As we discussed, some of the additionally retained testcases operated as stepping stones, letting it further explore states that eventually led to additional bugs. On the contrary, the queue median size growth for the all-or-nothing approach of context is on average 81.7% compared to lto, with peak values on grok (331.2%), libxml2 (200%), and njs (468.9%).

To better put these numbers in perspective, we first study how many executions each fuzzer completed in a unit of time. This metric is affected by the fuzzer's novelty search, as exploration can take different turns among different concepts: this impacts both the execution time of individual testcases (depending on the regions visited) and queue management (e.g., culling) costs when its size grows. Compared to the baseline lto, we observe for context a reduction of the fuzzing execution throughput by 20.3% on average, whereas predictive is slower than lto by only **6.5%** on average[8].

[7]The reader may wonder why lto finds bugs in well-tested software. While OSS-Fuzz conducts daily 5h tests on them, we believe that, as studied in [14], our collision-free configuration is more performant thanks to LTO effects.

[8]As a reference, this overhead is lower than with other compile-time transformations often employed to improve fuzzing effectiveness, such as multi-byte comparison splitting to bypass magic value checks [59].

TABLE IV. MEDIAN QUEUE SIZE AND EXECUTIONS/SECOND RATIO FOR EACH FUZZER ACROSS 20 TRIALS OF THE FUZZBENCH PROGRAMS.

	Queue size				Executions per second			
Benchmark	**context**	**predictive**	**lto**	**random**	**context**	**predictive**	**lto**	**random**
`ffmpeg`	11202	9787	9713	8711	148	189	190	143
`file`	4046	2681	1734	2645	425	463	501	425
`grok`	17651	4503	4093	4975	35	152	131	137
`libarchive`	8007	6938	5526	5410	1659	1421	1601	1500
`libgit2`	2443	1190	1128	1271	899	826	868	931
`libhevc`	13229	8727	7515	11161	166	153	122	172
`libhtp`	9243	13878	6466	13990	2193	2881	2274	2814
`libxml2`	41928	15652	13977	15126	1352	1090	1132	1249
`matio`	15040	10374	9068	10935	298	415	492	334
`muparser`	1837	1522	1184	1828	1215	2183	3221	2108
`ndpi`	1623	1673	1651	1783	38	39	42	42
`njs`	27660	5083	4862	5236	498	570	650	623
`openh264`	6904	8031	5239	7770	6	8	10	9
`stb`	3761	6297	3228	6102	1414	1384	1565	1330
`usrsctp`	2632	1700	1631	1635	2351	2164	2318	2136
`zstd`	26711	25671	18464	28259	6516	5149	6307	4841
Geo-mean	7784	5416	4283	5528	431	506	541	502

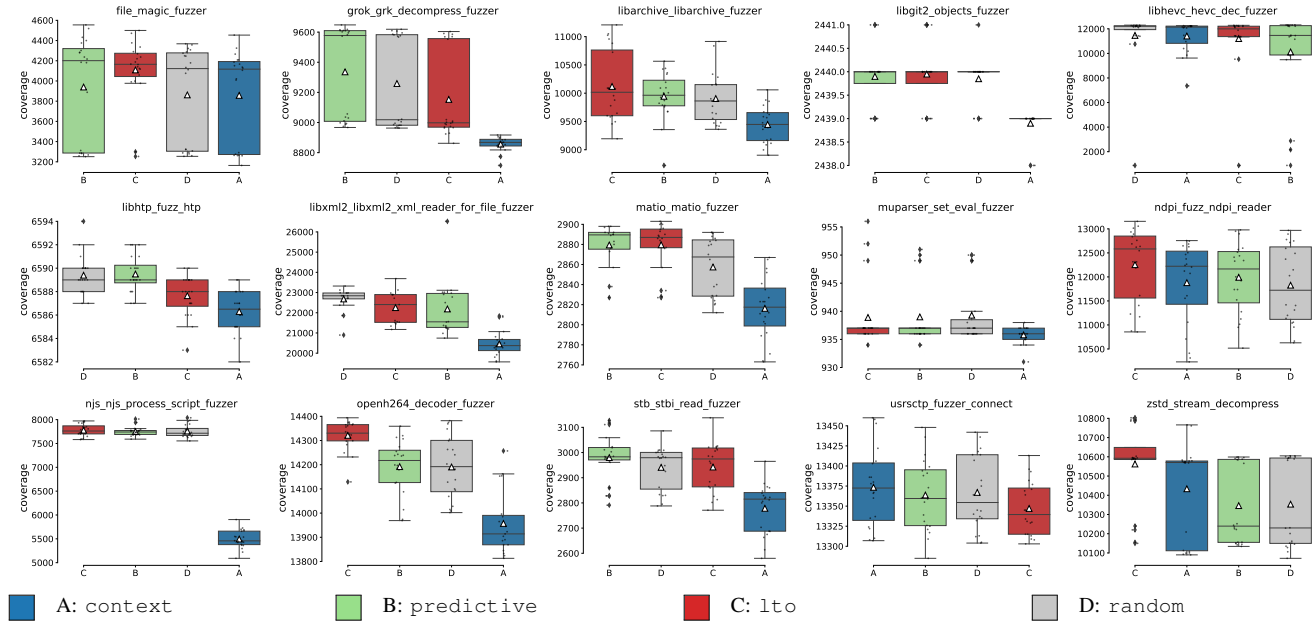

Fig. 4. Boxplots for FuzzBench programs for the median number of discovered control-flow edges on 20 trials. Symbols and ordering are as in Figure 2. Coverage data for `ffmpeg` is not available due to a known bug in the FuzzBench coverage measurer.

Next, we study how code coverage is affected. We recall that, unlike fuzzing techniques that aim to increase coverage (for example, to help a fuzzer meet checksums or other structural input constraints [15]), context-sensitivity aims to improve the exploitation stage (Section III). Both `predictive` and `context` favor a more pervasive analysis of program states already within reach of a CGF system based on edge coverage: hence, one cannot expect appreciable code coverage improvements from either. However, if a context-sensitive fuzzer faces significant internal wastage, this will be reflected in an appreciably lower coverage: with more testcases to mutate and/or lower throughput, the fuzzer will fall short of the time needed to schedule or sufficiently mutate testcases that lead to more coverage. As Figure 4 shows, compared to `lto`, the best-effort approach of `context` is appreciably detrimental for code coverage on several benchmarks (`libarchive`, `libhtp`, `libxml2`, `matio`, `njs`, `openh64`, `stb`).

Our predictive approach performs surprisingly well, obtaining coverage close to `lto` on all subjects except `libhevc`, `ndpi`, `openh264`, and `ztd` and even improving it for 8 subjects, showing almost no internal wastage. As we discussed in Section VI-A2 for the bugs found only by `predictive`, we attribute this improvement to how the refined data-flow along cloned call sites may occasionally help the fuzzer retain and later mutate testcases that eventually unlock new code.

C. RQ3: Analysis and Compilation Costs

Our approach incurs direct and indirect preparation costs for the transformed program that we feed to a fuzzer.

TABLE V. POINTS-TO COSTS AND STATISTICS FOR OUR SUBJECTS.

Benchmark	Time (s)	Memory (MB)	|Set| (avg.)	|Set| (σ)
ffmpeg	2193.75	22553	4.38	68.31
file	25.07	522	2.79	6.26
grok	181.55	2797	1.74	2.92
libarchive	103.5	2073	2.1	15.21
libgit2	446.52	4711	2.41	9.91
libhevc	93.73	3143	1.15	0.52
libhtp	161.98	647	56.1	43.79
libxml2	648.34	4289	9.83	12.26
matio	111.34	1181	32.47	30.76
muparser	12.53	487	2.9	5.83
ndpi	265.14	3250	115.5	87.18
njs	185.14	2012	16.22	11.74
openh264	108.55	2269	2.11	13.39
stb	12.86	413	2.08	2.28
usrsctp	1095.52	5851	55.85	33.07
zstd	45.47	1206	13.85	5.25
Geo-mean	139.94	2007.8	7.09	11.074

Direct costs include generating clones (typically a very fast operation) and running the analyses behind our predictive policy. Table V reports the CPU time and memory usage from running points-to analysis on each whole-program IR file. SVF takes on average 139.94 seconds and 1.96 GB of memory to analyze a program, peaking at 2193.75 seconds and 22 GB on a larger subject like ffmpeg. Memory usage is < 6 GB for all but one subject. Table V reports the average number of possible pointed abstract objects per call-site argument.

Indirect costs for IR preparation include the impact of cloning on the binary compilation process orchestrated by the off-the-shelf fuzzer. For predictive, compilation time increased on average by 153 seconds (peaking at 443 on ffmpeg). As for the resulting binary size increase, which includes the fuzzer's instrumentation for context-sensitive edge instances, we observe a geometric mean of 3.6x and a peak value of 10.1x (stb grows from 1.45 MB to 14.8 MB), while libarchive sees the largest produced binary with its 46 MB (initial size: 34.1 MB). In the context of fuzzing, though, such increases hardly affect performance. This applies to both persistent fuzzing scenarios (as with FuzzBench), where a binary is (re)loaded in memory only sporadically, and fork-based settings, which benefit from copy-on-write OS mechanisms. This observation is backed by the experiments of Section VI-B: in our tests, trading such additional space for supporting selective collision-free context-sensitivity did not harm the execution speed and effectiveness of our fuzzers.

D. Discussion

In our tests, predictive context-sensitive fuzzing significantly outperforms the all-or-nothing, best-effort approach pioneered in ANGORA. The internal wastage effects that characterize the latter make it fall behind even the random fuzzer in several tested dimensions (e.g., 110 vs 102 unique bugs in RQ1). Our data flow-based predictive policy largely outperforms random and the three topological policies discussed in Section IV-B that we evaluated in early tests.

The results of our investigation back the expectation that providing efficient and collision-free context-sensitivity only for the callees that matter—heuristically identified with a predictor based on diverse incoming data-flows at call sites—offers a practical, cost-effective, and scalable way for context-sensitive coverage-guided fuzzing. Detailed analyses of the identified bugs revealed that not only our fuzzers found *more* and *different* bugs than both context and lto, but also uncovered enduring issues in well-tested subjects.

Our evaluation is centered on bug finding results since, as repeatedly evidenced in the literature (e.g., most recently in [60]), coverage-based benchmarking is only one part of the story and, in our scenario, the goal is to better exploit coverage already reached by context-insensitive solutions. Simply reaching some code is often insufficient to trigger a bug, as the execution of a certain sequence of statements may be needed to enter the "right" state [60].

Our technique comes with tenable one-time compilation and analysis costs, limited run-time overhead, and no queue explosion effects. The moderate number of additionally retained testcases was instrumental for discovering more bugs in the tested benchmarks. Also, some of them occasionally helped the fuzzer reach new code through subsequent mutations.

VII. RELATED WORKS

Local Feedbacks: A few function-local feedbacks have been proposed as a replacement or extension of code coverage.

CollAFL [16] analyzes the detrimental effects of collisions in AFL-style edge coverage tracking, proposing a method to reduce them. The technique based on breaking critical edges that is adopted by AFL++ and LIBFUZZER and we use in this paper fully removes them. The work also argues that tracking full paths (vs. edges) is infeasible in practice. PathAFL [61] hashes whole-execution paths with pruning heuristics, but the approach has yet to gain traction in the fuzzing community.

Padhye et al. [62] analyze alternatives such as the number of bits matched between operands of integer comparisons (for input-dependent conditions that are difficult to meet) or the size of allocation operations (for memory corruption bugs). Wang et al. [10] study, among others, extensions for edge coverage as a feedback: for instance, they evaluate an *n-gram* feedback to track bounded-length sequences of consecutively traversed edges as a better approximation of the program behaviors.

Finally, other efforts investigate auxiliary feedbacks involving data profiles [58], [49], [63]. As one may naturally augment local feedbacks with our cloning-based context-sensitivity, future research may involve identifying profitable combinations.

Calling Contexts: Programming language literature largely studied calling contexts and their portions (e.g., [36], [64], [65], [38]). Due to their sheer number, a static enumeration of calling contexts is often unfeasible [65], and even space-efficient dynamic methods need wide identifiers to keep collisions low [66]. Furthermore, for complex programs, short executions often result in dozens of million distinct contexts [13], [67]. Unlike cloning, these techniques incur non-negligible temporal or spatial overheads, hindering an effective composition with local feedbacks used by fuzzers. Also, we have shown that full context-sensitivity—unlike selectivity—can be unnecessarily inefficient when fuzzing. Other works [68], [69] study calling contexts that, if seen as distinct, would bring more accurate points-to sets. While

better sets may refine our data flow-based priority values, the selectivity choices from these works would hardly help a fuzzer on their own, as they optimize for a different goal.

A particularly relevant work in the field is [11], as it pioneered the concept of cloning for static analysis. The authors simulate the creation of clones for all the call-graph acyclic paths reaching each function and use a context numbering scheme that exposes commonalities in context-sensitive relations, enabling their efficient encoding through ordered binary decision diagrams. They then store all program information and results as relations and encode program analyses as Datalog operations. The work showcases scalable implementations of context-sensitive points-to and other analyses on Java code.

Recently, some works leveraged calling contexts in special-purpose fuzzers. FIFUZZ [70] targets bugs in error handling code, which may trigger only when the error site fails in a specific calling context. CONZZER [71] focuses on data races occurring in specific runtime contexts, modeling execution contexts through a concurrency coverage metric that describes thread interleavings with runtime calling contexts. It would be interesting to explore synergies between these methods and what we propose in this paper for general fuzzing systems.

Directed Fuzzing: While directed fuzzing is a long-studied subject [72], [73], its combination with grey-box fuzzing is more recent [20]. This fuzzing flavor can guide execution towards specific program points deemed interesting: for instance, a crash site from a core dump. AFLGO [20] builds a whole-program inter-procedural control flow graph (iCFG) and assigns weights to basic blocks to define a distance function from the entry point to a target location. When fuzzing, it gradually assigns more energy to testcases that are closer to the target locations. HAWKEYE [30] improves, among others, the iCFG construction by reasoning on indirect-call targets obtained from a points-to analysis. While directed fuzzing focuses on reaching predetermined locations based on user-specified criteria, our approach automatically selects interesting program points for context-sensitive coverage tracking. However, our approach may potentially enhance directed fuzzing in two ways: (i) context-sensitivity may refine some points-to sets during iCFG construction and (ii) given a stack-trace, we may clone only the context that leads to the target program state and assign ad-hoc weights to clones.

Software Hardening: A few hardening solutions resort to cloning techniques, often in combination with points-to analyses. Constantine [74] uses function cloning to improve the accuracy of points-to analysis by adding context-sensitivity. The authors apply it to the cryptographic functions in a library that are secret-sensitive: as those are typically in limited number, this somewhat bounds explosion issues. ProbeGuard [75] clones functions to provide hardened versions that can be hotpatched to protect programs from probing attacks. Control-flow integrity solutions leverage type or pointer analyses to enumerate the possible targets of an indirect branch, and restrict the code to follow one of them [76], [77], [78]. DynPTA [32] enhances a Steensgaard-style points-to analysis with context-sensitive heap modeling using function summaries to distinguish different allocation sites, treating them as virtual clones of the original function.

VIII. CONCLUDING REMARKS

We presented a novel approach to context-sensitivity in fuzzing, terming it predictive. Our proposal stems from the analysis of existing context-sensitive approaches, which track full calling contexts and allow context/edge hash collisions for the sake of a practical implementation. Such approaches face an impossible trade-off: either allow too many collisions and lose context (but also path[1]) sensitivity, or allow too few and incur trashing behavior due to queue/map explosion. With our approach, we show that a profitable avenue exists if we proactively select (and clone) only the contexts that look more promising as predicted by a program analysis oracle, forbidding hash collisions and avoiding internal wastage. Our tests show that data-flow diversity can serve as one effective predictor for such contexts, with significant improvements compared to the state of the art (e.g., +22.55% total bugs on ANGORA-style context-sensitivity). They also found 8 enduring security issues in 5 well-tested programs, with 6 CVE IDs issued.

ACKNOWLEDGEMENTS

We would like to thank the reviewers for their feedback. We are also grateful to Mathias Payer for his comments on a prior version of this work and to Giacomo Priamo and Slasti Mormanti for their valuable suggestions when finalizing the manuscript. This work was supported by the Italian MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU through project SERICS (PE00000014), by the Dutch Ministry of Economic Affairs and Climate Policy (EZK) through the AVR "Memo" project, by the Dutch Research Council (NWO) through project "INTERSECT", and by the European Union's Horizon Europe programme under grant agreement No. 101120962 ("Rescale").

REFERENCES

[1] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 711–725.

[2] "Google OSS-Fuzz: continuous fuzzing of open source software," https://github.com/google/oss-fuzz, 2016, [Online; accessed 28 Mar. 2023].

[3] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "ParmeSan: Sanitizer-guided Greybox Fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2289–2306. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/osterlund

[4] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: fuzzing with input-to-state correspondence," in *26th Annual Network and Distributed System Security Symposium, NDSS*, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/

[5] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym: A practical concolic execution engine tailored for hybrid fuzzing," in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. SEC'18. USENIX Association, 2018, pp. 745–761.

[6] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

[7] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. ACM, 2014, p. 435–445. [Online]. Available: https://doi.org/10.1145/2568225.2568271

[8] H. Zheng, J. Zhang, Y. Huang, Z. Ren, H. Wang, C. Cao, Y. Zhang, F. Toffalini, and M. Payer, "FishFuzz: Throwing larger nets to catch deeper bugs," in *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Aug. 2023.

[9] V. J. M. Manès, S. Kim, and S. K. Cha, "Ankou: Guiding grey-box fuzzing towards combinatorial difference," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. ACM, 2020, pp. 1024–1036. [Online]. Available: https://doi.org/10.1145/3377811.3380421

[10] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. USENIX Association, Sep. 2019, pp. 1–15. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/wang

[11] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," *SIGPLAN Not.*, vol. 39, no. 6, pp. 131–144, Jun. 2004. [Online]. Available: https://doi.org/10.1145/996893.996859

[12] K. Hazelwood and D. Grove, "Adaptive online context-sensitive inlining," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '03. IEEE Computer Society, 2003, pp. 253–264.

[13] D. C. D'Elia, C. Demetrescu, and I. Finocchi, "Mining hot calling contexts in small space," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. ACM, 2011, pp. 516–527. [Online]. Available: https://doi.org/10.1145/1993498.1993559

[14] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[15] A. Fioraldi, D. C. D'Elia, and E. Coppa, "WEIZZ: Automatic grey-box fuzzing for structured binary formats," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. ACM, 2020. [Online]. Available: https://doi.org/10.1145/3395363.3397372

[16] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 679–696.

[17] J. Metzman, L. Szekeres, L. M. R. Simon, R. T. Sprabery, and A. Arya, "Fuzzbench: An open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, New York, NY, USA, 2021.

[18] M. Payer, "The fuzzing hype-train: How random testing triggers thousands of crashes," *IEEE Security and Privacy*, vol. 17, no. 1, pp. 78–82, 2019.

[19] M. Böhme and S. Paul, "A probabilistic analysis of the efficiency of automated software testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 4, pp. 345–360, 2016.

[20] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. ACM, 2017, pp. 2329–2344. [Online]. Available: https://doi.org/10.1145/3133956.3134020

[21] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. ACM, 2011, p. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[22] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, Aug. 2012, pp. 445–458. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[23] S. Poeplau and A. Francillon, "Symbolic execution with symcc: Don't interpret, compile!" in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 181–198. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[24] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the Network and Distributed System Security Symposium*, ser. NDSS'08, 2008.

[25] M. Zalewski, "American Fuzzy Lop - Whitepaper," https://lcamtuf.coredump.cx/afl/technical_details.txt, 2016, [Online; accessed 28 Mar. 2023].

[26] LLVM Project, "libFuzzer – a library for coverage-guided fuzz testing." https://llvm.org/docs/LibFuzzer.html, Sep. 2018, [Online; accessed 28 Mar. 2023].

[27] J. D. DeMott and R. Enbody, "Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing," ser. Black Hat USA, 2007.

[28] LLVM, "SanitizerCoverage - Edge coverage," https://clang.llvm.org/docs/SanitizerCoverage.html#edge-coverage, 2021, [Online; accessed 28 Mar. 2023].

[29] M. Hind, "Pointer analysis: Haven't we solved this problem yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '01. ACM, 2001, p. 54–61. [Online]. Available: https://doi.org/10.1145/379605.379665

[30] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. ACM, 2018, p. 2095–2108. [Online]. Available: https://doi.org/10.1145/3243734.3243849

[31] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. ACM, 2018, p. 1470–1486. [Online]. Available: https://doi.org/10.1145/3243734.3243797

[32] T. Palit, J. Firose Moon, F. Monrose, and M. Polychronakis, "Dynpta: Combining static and dynamic analysis for practical selective data protection," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1919–1937.

[33] Y. Sui and J. Xue, "Svf: Interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. ACM, 2016, p. 265–266. [Online]. Available: https://doi.org/10.1145/2892208.2892235

[34] J. Kuderski, J. A. Navas, and A. Gurfinkel, "Unification-based pointer analysis without oversharing," in *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*, C. W. Barrett and J. Yang, Eds. IEEE, 2019, pp. 37–45. [Online]. Available: https://doi.org/10.23919/FMCAD.2019.8894275

[35] T. Yu and O. Kaser, "A note on "on the conversion of indirect to direct recursion"," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 6, pp. 1085–1087, Nov. 1997. [Online]. Available: https://doi.org/10.1145/267959.269973

[36] O. G. Shivers, "Control-flow analysis of higher-order languages of taming lambda," Ph.D. dissertation, 1991, uMI Order No. GAX91-26964.

[37] E. M. Nystrom, H.-S. Kim, and W.-m. W. Hwu, "Bottom-up and top-down context-sensitive summary-based pointer analysis," in *Static Analysis*, R. Giacobazzi, Ed. Springer Berlin Heidelberg, 2004, pp. 165–180.

[38] G. Ausiello, C. Demetrescu, I. Finocchi, and D. Firmani, "K-calling context profiling," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12. ACM, 2012, pp. 867–878. [Online]. Available: https://doi.org/10.1145/2384616.2384679

[39] Z. Y. Ding and C. L. Goues, "An empirical study of oss-fuzz bugs," *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 131–142, 2021.

[40] Y. Jeon, W. Han, N. Burow, and M. Payer, "Fuzzan: Efficient sanitizer metadata design for fuzzing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 249–263. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/jeon

[41] W. H. Harrison, "Compiler analysis of the value ranges for variables," *IEEE Transactions on Software Engineering*, vol. 3, no. 03, pp. 243–250, may 1977.

[42] I. A. Mason, "Whole Program LLVM in Go," https://github.com/SRI-CSL/gllvm, 2021, [Online; accessed 2 Sep. 2021].

[43] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1497–1511.

[44] I. Baev and Q. I. Center, "Profile-based indirect call promotion," in *LLVM Developers Meeting, Oct*, 2015.

[45] N. Amit, F. Jacobs, and M. Wei, "Jumpswitches: Restoring the performance of indirect branches in the era of spectre," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Jul. 2019, pp. 285–300. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/amit

[46] V. Duta, C. Giuffrida, H. Bos, and E. van der Kouwe, "Pibe: Practical kernel control-flow hardening with profile-guided indirect branch elimination," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021. ACM, 2021, p. 743–757. [Online]. Available: https://doi.org/10.1145/3445814.3446740

[47] P. Biswas, N. Burow, and M. Payer, "Code specialization through dynamic feature observation," in *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '21. ACM, 2021, p. 257–268. [Online]. Available: https://doi.org/10.1145/3422337.3447844

[48] K. Lu, "Practical program modularization with type-based dependence analysis," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, may 2023, pp. 1610–1624. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00092

[49] A. Mantovani, A. Fioraldi, and D. Balzarotti, "Fuzzing with data dependency information," in *7th IEEE European Symposium on Security and Privacy*, ser. EuroS&P '22, IEEE, Ed., 2022.

[50] A. Fioraldi, A. Mantovani, D. Maier, and D. Balzarotti, "Dissecting American Fuzzy Lop: A FuzzBench evaluation," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 2, mar 2023. [Online]. Available: https://doi.org/10.1145/3580596

[51] D. Liu, J. Metzman, M. Böhme, O. Chang, and A. Arya, "SBFT Tool Competition 2023 - Fuzzing Track," in *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, 2023, pp. 51–54.

[52] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "SoK: Sanitizing for security," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1275–1295.

[53] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. ACM, 2018, pp. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[54] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, "Towards optimization-safe systems: Analyzing the impact of undefined behavior," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. ACM, 2013, p. 260–275. [Online]. Available: https://doi.org/10.1145/2517349.2522728

[55] C. Salls, C. Jindal, J. Corina, C. Kruegel, and G. Vigna, "Token-Level fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2795–2809. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/salls

[56] E. Güler, P. Görz, E. Geretto, A. Jemmett, S. Österlund, H. Bos, C. Giuffrida, and T. Holz, "Cupid: Automatic fuzzer selection for collaborative fuzzing," in *Annual Computer Security Applications Conference*, ser. ACSAC '20. ACM, 2020, pp. 360–372. [Online]. Available: https://doi.org/10.1145/3427228.3427266

[57] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Aug. 2019, pp. 1967–1983. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/chen-yuanliang

[58] A. Fioraldi, D. C. D'Elia, and D. Balzarotti, "The use of likely invariants as feedback for fuzzers," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2829–2846. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/fioraldi

[59] "Circumventing Fuzzing Roadblocks with Compiler Transformations," https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/, 2016, [Online; accessed 28 Mar. 2023].

[60] M. Böhme, L. Szekeres, and J. Metzman, "On the reliability of coverage-based fuzzer benchmarking," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. ACM, 2022, pp. 1621–1633. [Online]. Available: https://doi.org/10.1145/3510003.3510230

[61] S. Yan, C. Wu, H. Li, W. Shao, and C. Jia, "Pathafl: Path-coverage assisted fuzzing," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '20. ACM, 2020, pp. 598–609. [Online]. Available: https://doi.org/10.1145/3320269.3384736

[62] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar, "FuzzFactory: Domain-specific fuzzing with waypoints," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360600

[63] A. Herrera, M. Payer, and A. Hosking, "datAFLow: Towards a data-flow-guided fuzzer," in *1st International Fuzzing Workshop*, ser. FUZZING '22, I. Society, Ed., 2022.

[64] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, ser. PLDI '97. ACM, 1997, pp. 85–96. [Online]. Available: https://doi.org/10.1145/258915.258924

[65] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang, "Precise calling context encoding," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. ACM, 2010, pp. 525–534. [Online]. Available: https://doi.org/10.1145/1806799.1806875

[66] M. D. Bond and K. S. McKinley, "Probabilistic calling context," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, ser. OOPSLA '07. ACM, 2007, pp. 97–112. [Online]. Available: https://doi.org/10.1145/1297027.1297035

[67] D. C. D'Elia, C. Demetrescu, and I. Finocchi, "Mining hot calling contexts in small space," *Software: Practice and Experience*, vol. 46, no. 8, pp. 1131–1152, 2016. [Online]. Available: https://doi.org/10.1002/spe.2348

[68] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "A principled approach to selective context sensitivity for pointer analysis," *ACM Trans. Program. Lang. Syst.*, vol. 42, no. 2, may 2020. [Online]. Available: https://doi.org/10.1145/3381915

[69] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "Introspective analysis: Context-sensitivity, across the board," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. ACM, 2014, pp. 485–495. [Online]. Available: https://doi.org/10.1145/2594291.2594320

[70] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu, "Fuzzing error handling code using Context-Sensitive software fault injection," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2595–2612. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/jiang

[71] Z. Jiang, J. Bai, K. Lu, and S. Hu, "Context-sensitive and directional concurrency fuzzing for data-race detection," in *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022.

[72] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. ACM, 2005, pp. 213–223. [Online]. Available: https://doi.org/10.1145/1065010.1065036

[73] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 474–484.

[74] P. Borrello, D. C. D'Elia, L. Querzoni, and C. Giuffrida, "Constantine: Automatic side-channel resistance using efficient control and data flow linearization," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. ACM, 2021.

[75] K. Bhat, E. van der Kouwe, H. Bos, and C. Giuffrida, "Probeguard: Mitigating probing attacks through reactive program transformations," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. ACM, 2019, pp. 545–558. [Online]. Available: https://doi.org/10.1145/3297858.3304073

[76] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing Forward-Edge Control-Flow Integrity in GCC and LLVM," in *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2014.

[77] Clang, "LLVM's Control Flow Integrity," 2018, [Online; accessed 28 Mar. 2023]. [Online]. Available: https://clang.llvm.org/docs/ControlFlowIntegrity.html

[78] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive cfi," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. ACM, 2015, p. 927–940. [Online]. Available: https://doi.org/10.1145/2810103.2813673

APPENDIX A
ADDITIONAL BUG ANALYSIS RESULTS

The CVE identifiers for the security issues in the FuzzBench programs mentioned in Section VI-A3 are the following: CVE-2022-28041, CVE-2022-28042, and CVE-2022-28048 for `stb`, CVE-2022-1475 for `ffmpeg`, CVE-2022-1515 for `matio`, and CVE-2022-28049 for `njs`.

This appendix includes four tables (Table VI, VII, VIII, and IX) that complement the bug counts reported in Table III and the inclusion relations of Figure 3 with more detailed comparisons based on bug identity at each benchmark.

Benchmark	Only `predictive`	Both	Only `context`
ffmpeg	5	6	0
file	1	2	1
grok	5	2	0
libarchive	0	0	0
libgit2	0	3	0
libhevc	0	1	1
libhtp	0	5	0
libxml2	19	3	0
matio	0	26	17
muparser	1	0	0
ndp	6	11	1
njs	1	0	0
openh264	1	7	0
stb	3	15	0
usrsctp	0	0	0
zstd	1	1	0

TABLE VI. INCLUSION RELATIONS FOR BUGS FOUND BY `predictive` AND `lto` IN THE FUZZBENCH EXPERIMENTS (CF. LEFT PART OF FIGURE 3).

Benchmark	Only `predictive`	Both	Only `lto`
ffmpeg	2	9	1
file	2	1	0
grok	1	6	0
libarchive	0	0	2
libgit2	0	3	0
libhevc	0	1	1
libhtp	0	5	1
libxml2	6	16	0
matio	2	24	2
muparser	1	0	0
ndp	2	15	3
njs	0	1	0
openh264	1	7	1
stb	7	11	0
usrsctp	0	0	0
zstd	0	2	0

TABLE VII. INCLUSION RELATIONS FOR BUGS FOUND BY `predictive` AND `lto` IN THE FUZZBENCH EXPERIMENTS (CF. LEFT PART OF FIGURE 3).

Benchmark	Only `context`	Both	Only `lto`
ffmpeg	0	6	4
file	2	1	0
grok	0	2	4
libarchive	0	0	2
libgit2	0	3	0
libhevc	1	1	1
libhtp	0	5	1
libxml2	0	3	13
matio	17	26	0
muparser	0	0	0
ndp	2	10	8
njs	0	0	1
openh264	0	7	1
stb	4	11	0
usrsctp	0	0	0
zstd	0	1	1

TABLE VIII. INCLUSION RELATIONS FOR BUGS FOUND BY `context` AND `lto` IN THE FUZZBENCH EXPERIMENTS (CF. LEFT PART OF FIGURE 3).

Benchmark	Only `predictive`	All	Only others
ffmpeg	2	9	1
file	1	2	1
grok	1	6	0
libarchive	0	0	2
libgit2	0	3	0
libhevc	0	1	2
libhtp	0	5	1
libxml2	6	16	0
matio	0	26	17
muparser	1	0	0
ndp	1	16	4
njs	0	1	0
openh264	1	7	1
stb	3	15	0
usrsctp	0	0	0
zstd	0	2	0

TABLE IX. INCLUSION RELATIONS FOR BUGS FOUND BY `predictive` VS. THE ENSEMBLE OF `context` AND `lto` IN THE FUZZBENCH EXPERIMENTS (CF. LEFT PART OF FIGURE 3). NOTE THAT THE ENSEMBLE HAS THE UNFAIR ADVANTAGE OF HAVING DONE TWICE AS MANY TRIALS.